Software Engineering Institute

PSP_{VDC}: An Adaptation of the PSP that Incorporates Verified Design by Contract

Silvana Moreno, Universidad de la República
Álvaro Tasistro, Universidad ORT Uruguay
Diego Vallespir, Universidad de la República
William Nichols, Carnegie Mellon University

May 2013

TECHNICAL REPORT
CMU/SEI-2013-TR-005
ESC-TR-2013-005

Software Engineering Process Management

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

SEI Administrative Agent
AFLCMC/PZE
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0000372

Table of Contents

List of Figures

List of Tables

Acknowledgments

We gratefully thank our editor Erin Harper for helping us bring this report into presentable form, and for doing it on a tight schedule.

Abstract

The Personal Software Process (PSP) promotes the use of careful procedures during all stages of development with the aim of increasing an individual's productivity and producing high quality final products. Formal methods use the same methodological strategy as the PSP: emphasizing care in development procedures as opposed to relying on testing and debugging. They also establish the radical requirement of proving mathematically that the programs produced satisfy their specifications. Design by Contract (DbC) is a technique for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language. When appropriate techniques and tools are incorporated to prove that the components satisfy the established requirements, the method is called Verified Design by Contract (VDbC).

This paper describes a proposal for integrating VDbC into PSP in order to reduce the amount of defects present at the Unit Testing phase, while preserving or improving productivity. The resulting adaptation of the PSP, called PSP_{VDC}, incorporates new phases, modifies others, and adds new scripts and checklists to the infrastructure. Specifically, the phases of Formal Specification, Formal Specification Review, Formal Specification Compile, Test Case Construct, Pseudo Code, Pseudo Code Review, and Proof are added.

1 Introduction

Software increases in size and complexity each year and plays a larger role in many aspects of our lives. Because the development of software is a creative and intellectual activity performed by human beings, it is normal for the development team to make mistakes, both because of the complexity of the software and because of human nature itself. These mistakes often end up as defects in software products and can cause severe damage when the software is executed. Research on developing defect-free software has led to the development of many processes and methods that aim to detect defects before the product is delivered to the users.

The Personal Software Process (PSP) incorporates process discipline and quantitative management into the software engineer's individual development work. It promotes the exercise of careful procedures during all stages of development with the aim of increasing the individual's productivity and achieving high quality final products [Humphrey 2005, Humphrey 2006].

The PSP course progressively teaches engineers planning, development, and process assessment practices as they build actual programs. Performance data from students in this course has been collected and statistically analyzed, and the results show that PSP substantially reduces the amount of defects per lines of code that survive until the Unit Testing phase [Hayes 1997] [Rombach 2007], indicating that employment of PSP significantly improves product quality.

Still, removing defects at the Unit Testing phase costs five to seven times more than removing them in earlier phases of the PSP [Vallespir 2011, Vallespir 2012]. Because 38% of the injected defects are still present at Unit Testing, opportunities exist for improvement in the early detection of defects using TSP.

Formal methods use the same methodological strategy as the PSP: emphasizing care in development procedures as opposed to relying on testing and debugging. They also establish the radical requirement of proving mathematically that the programs produced satisfy their specifications. Design by Contract (DbC) is a technique devised and patented by Bertrand Meyer for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language [Meyer 1992]. The formal languages that are used for DbC are seamlessly integrated into the programming language to allow specified conditions to be evaluated at runtime, with violations of these conditions managed with exception handling. When appropriate techniques and tools are incorporated to prove that the components satisfy the established requirements, the method is called Verified Design by Contract (VDbC).

In this paper we propose a way to integrate VDbC into PSP to reduce the amount of defects present at the Unit Testing phase, while at the same time preserving or improving productivity. The resulting adaptation of the PSP, called PSP_{VDC}, incorporates new phases, modifies others, and adds new scripts and checklists to the infrastructure. Specifically, the phases of Formal Specification, Formal Specification Review, Formal Specification Compile, Test Case Construct, Pseudo Code, Pseudo Code Review, and Proof are added. At a later stage, controlled experiments will be conducted for obtaining results about the improvements achieved by our adaptation. We expect that such experiments will motivate further adjustments to the process so that it eventually becomes practical enough to be employed in industry.

We know of only three works in the literature that propose a combination of PSP and formal methods. Babar and Potter [Babar 2005] combine Abrial's B Method with PSP into B-PSP. They add the phases of Specification, Auto Prover, Animation, and Proof. A new set of defect types is added and logs are modified so as to incorporate data extracted from the B machine's structure. The goal of this work is to provide the individual B developers with a paradigm of measurement and evaluation that promotes reflection on the practice of the B method, inculcating the habit of recognizing causes of defects injected to help prevent them in the future. In comparison to B, our chosen formal method is significantly lighter and so, we expect, easier to incorporate into industrial practice.

Suzumori, Kaiya, and Kaijiri proposed the combination of VDM and PSP [Suzumori 2003]. In their method, the Design phase is modified incorporating the formal specification in the VDM-SL language. New phases are also added: VDM-SL Review, Syntax Check, Type Check, and Validation. A prototype course requiring each student to carry out nine exercises applying VDM on the PSP was designed. After this work was concluded, the research was discontinued for reasons internal to the organization.[1]

Contemporaneously to our work, Kusakabe, Omori, and Araki proposed combining PSP and VDM with the goal of avoiding the injection of defects at the design phase [Kusakabe 2012]. They use automated tools (VDMTools) for syntax checking, type checking, interpretation, and generation of proof obligations. For evaluating the resulting process they had an engineer apply ordinary PSP to the course materials of PSP for Engineers I, then apply the combination of PSP and VDM to a few exercises in the course material of PSP for Engineers II. The experimental results show a successful reduction of the number of defects, without decreased productivity. However, they note that proficiency in the programming language and software development skills might affect the results.

The rest of this paper describes the PSP and PSP_{VDC} methods and is structured as follows: Section 2 provides a general description of PSP, while Section 3 gives a general description of formal methods—VDbC in particular. Section 4 presents the adaptation of PSP to incorporate VDbC. Finally, Section 5 describes conclusions and further work.

[1] As communicated by the authors via e-mail.

2 Personal Software Process

The PSP was proposed in 1995 by Watts Humphrey at the Software Engineering Institute (SEI). It is aimed at increasing the quality of the products manufactured by individual professionals by improving their personal methods of software development. It takes into account diverse aspects of the software process, including planning, quality control, cost estimation, and productivity.

The PSP is divided into phases, as shown in Figure 1. A project begins with the requirements for a software module and ends when the software is released. The phases are: Planning, Design, Design Review, Code, Code Review, Compile, Unit Test, and Postmortem.

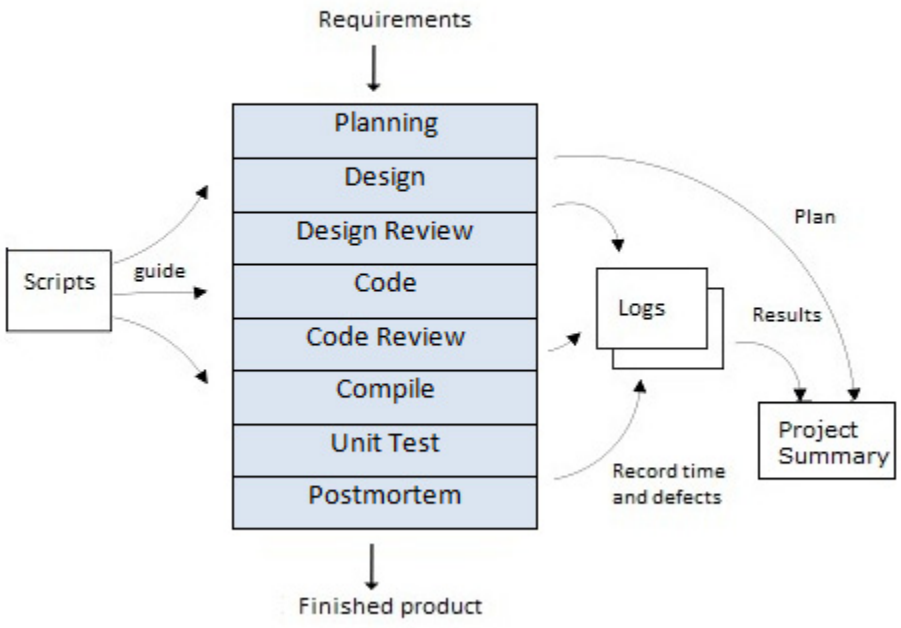

Figure 1: Phases of the PSP

In the PSP, all tasks and activities to be performed during software development are defined in a set of documents called scripts. Scripts dictate the course of the work and are to be followed in a disciplined manner. They also facilitate the collection of data about the software process, including time spent at each phase, defects detected at each phase, time spent in detection and correction, the phase at which each defect is detected and removed, and the classification of defects into types. This data is collected into logs and used to evaluate the quality of the process through the employment of indicators like defect density, review rate, and yield. All these measurements render a highly instrumented process, which is ideal for the realization of empirical studies [Wohlin 00]. The scripts used in PSP include the Process Script, Planning Script, Development Script, Design Review Script, Code Review Script, and Postmortem Script. Every script is composed of a purpose, a set of entry criteria, the activities to perform, and the expected outcomes (i.e., exit criteria).

The Process Script, shown in Table 1, contains a general program for the activities of Planning, Development, and Postmortem. The Development activity, in turn, consists of the phases Design, Design Review, Code, Code Review, Compile, and Unit Testing. Therefore, the Process Script describes the whole process.

Table 1: Process Script

Process Script

Purpose	To guide the development of module-level programs
Entry Criteria	- Problem description - PSP Project Plan Summary form - Size Estimating template - Historical size and time data (estimated and actual) - Time and Defect Recording logs - Defect Type, Coding, and Size Counting standards - Stopwatch (optional)

Step	**Activities**	**Description**
1	Planning	- Produce or obtain a requirements statement. - Use the PROBE method to estimate the added and modified size and the size prediction interval of this program. - Complete the Size Estimating template. - Use the PROBE method to estimate the required development time and the time prediction interval. - Complete a Task Planning template. - Complete a Schedule Planning template. - Enter the plan data in the Project Plan Summary form. - Complete the Time Recording log.
2	Development	- Design the program. - Document the design in the design templates. - Review the design and fix and log all defects found. - Implement the design. - Review the code and fix and log all defects found. - Compile the program and fix and log all defects found. - Test the program and fix and log all defects found. - Complete the Time Recording log.
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data.

Exit Criteria	- A thoroughly tested program - Completed Project Plan Summary form with estimated and actual data - Completed Size Estimating and Task and Schedule Planning templates - Completed Design templates - Completed Design Review and Code Review checklists - Completed Test Report template - Completed Process Improvement Proposal (PIP) forms - Completed Time and Defect Recording logs

The Planning Script, shown in Table 2, describes the Planning Phase. The goals of this phase are to arrive at a precise definition of the product to be constructed, estimate its size, and, on the basis of

personal productivity, estimate the time required for construction. As a method of estimation, PSP uses PROxy Based Estimation (PROBE) [Humphrey 05], which, by employing linear regression on historical data, yields an estimated size in lines of code (LOCs) and estimated time in minutes.

Table 2: Planning Script

Planning Script

Purpose	To guide the PSP planning process
Entry Criteria	- Problem description - PSP Project Plan Summary form - Size Estimating, Task Planning, and Schedule Planning templates - Historical size and time data (estimated and actual) - Time Recording log

Step	**Activities**	**Description**
1	Program Requirements	- Produce or obtain a requirements statement for the program. - Ensure that the requirements statement is clear and unambiguous. - Resolve any questions.
2	Size Estimate	- Produce a program conceptual design. - Use the PROBE method to estimate the added and modified size of this program. - Complete the Size Estimating template and Project Plan Summary form. - Calculate the 70% size prediction interval. (Note: This step is completed in the SEI student workbook.)
3	Resource Estimate	- Use the PROBE method to estimate the time required to develop this program. - Calculate the 70% size prediction interval. (Note: This step is completed in the SEI student workbook) - Using the "to-date %" from the most recently developed program as a guide, distribute the development time over the planned project phases. (Note: This step is completed in the SEI student workbook.)
4	Task and Schedule Planning	For projects lasting several days or more, complete the Task Planning and Schedule Planning templates.
5	Defect Estimate	- Based on your to-date data on defects per added and modified size unit, estimate the total defects to be found in this program. - Based on your "to-date %" data, estimate the number of defects to be injected and removed by phase.

Exit Criteria	- Documented requirements statement - Program conceptual design - Completed Size Estimating template - For projects lasting several days or more, completed Task and Schedule Planning templates - Completed Project Plan Summary form with estimated program size, development time, and defect data, and the time and size prediction intervals - Completed Time Recording log

The Development Script, shown in Table 3, describes the activities to be carried out at the phases of Design, Design Review, Coding, Code Review, Compilation, and Unit Test.

The Design phase consists of designing the program in a complete and unambiguous manner. PSP makes use of four templates to provide documentation of the design in four dimensions: static, dynamic, internal, and external. In particular, the operational specification template describes the interaction between user and system (i.e., the dynamic-external view). The functional specification template allows the definition of the structural features to be provided by the software product, among them classes and inheritance, externally visible attributes, and relations to other classes or parts (i.e., the dynamic-external and static-external views). The state specification template describes the set of states of the program, the transitions between states, and the actions to be taken at each transition (i.e., the dynamic-internal view). Finally, the logic template specifies the internal logic of the program (i.e., the static-internal view) in a concise and convenient way. Pseudo code is appropriate for this task.

Once the design is completed, PSP proceeds to the Design Review phase, described in the Design Review Script in Table 4. Reviews allow you to find defects prior to the first compilation or test. The Design Review phase includes the following checks, among others: that all requirements are taken into account, that the flow and structure of the program are adequate, and that methods and variables are used correctly.

During the Code phase the program is constructed, employing a programming language and a coding standard.

After this phase, a review of the code is carried out, making use of the Code Review Script shown in Table 5. Code review is a very effective and inexpensive method for finding defects [Hayes 1997, Vallespir 2012]. Both design and code reviews are carried out with the use of checklists, which are created and maintained by each individual engineer taking into account the defects that he/she usually introduces.

After Code Review is the Compile phase, which is the translation of the source program into machine language using a compiler. The phase involves correcting defects detected by the compiler.

The Unit Test phase consists of the execution of the test cases specified during the Design phase. The defects detected at Unit Test allow the quality of the product to be assessed. In PSP, a program is considered to be of adequate quality if it contains 5 or fewer defects per KLOC at Unit Test.

Table 3: Development Script

Development Script

Purpose	To guide the development of small programs
Entry Criteria	- Requirements statement - Project Plan Summary form with estimated program size and development time - For projects lasting several days or more, completed Task Planning and Schedule Planning templates - Time and Defect Recording logs - Defect Type standard and Coding standard

Step	**Activities**	**Description**
1	Design	- Review the requirements and produce an external specification to meet them. - Complete Functional and Operational Specification templates to record this specification. - Produce a design to meet this specification. - Record the design in Functional, Operational, State, and Logic Specification templates. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
2	Design Review	- Follow the Design Review script and checklist and review the design. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
3	Code	- Implement the design following the Coding standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
4	Code Review	- Follow the Code Review script and checklist and review the code. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
5	Compile	- Compile the program until there are no compile errors. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log. - Complete a Test Report template on the tests conducted and the results obtained.

Exit Criteria	- A thoroughly tested program that conforms to the Coding standard - Completed Design templates - Completed Design Review and Code Review checklists - Completed Test Report template - Completed Time and Defect Recording logs

Table 4: Design Review Script

Design Review Script

Purpose	To guide you in reviewing detailed designs
Entry Criteria	- Completed program design documented with the PSP Design templates - Design Review checklist - Design standard - Defect Type standard - Time and Defect Recording logs
General	Where the design was previously verified, check that the analyses: - covered all of the design - were updated for all design changes - are correct - are clear and complete

Step	**Activities**	**Description**
1	Preparation	- Examine the program and checklist and decide on a review strategy. - Examine the program to identify its state machines, internal loops, and variable and system limits. - Use a trace table or other analytical method to verify the correctness of the design.
2	Review	- Follow the Design Review checklist. - Review the entire program for each checklist category; do not try to review for more than one category at a time! - Check off each item as you complete it. - Complete a separate checklist for each product or product segment reviewed.
3	Fix Check	- Check each defect fix for correctness. - Re-review all changes. - Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space.

Exit Criteria	- A fully reviewed detailed design - One or more Design Review checklists for every design reviewed - Documented design analysis results - All identified defects fixed and all fixes checked - Completed Time and Defect Recording logs

Table 5: Code Review Script

Code Review Script

Purpose	To guide you in reviewing programs
Entry Criteria	- A completed and reviewed program design - Source program listing - Code Review checklist - Coding standard - Defect Type standard - Time and Defect Recording logs
General	Do the code review with a source-code listing; do not review on the screen!

Step	Activities	Description
1	Review	- Follow the Code Review checklist. - Review the entire program for each checklist category; do not try to review for more than one category at a time! - Check off each item as it is completed. - For multiple procedures or programs, complete a separate checklist for each.
2	Correct	- Correct all defects. - If the correction cannot be completed, abort the review and return to the prior process phase. - To facilitate defect analysis, record all of the data specified in the Defect Recording log instructions for every defect.
3	Check	- Check each defect fix for correctness. - Re-review all design changes. - Record any fix defects as new defects and, where you know the number of the defect with the incorrect fix, enter it in the fix defect space.

Exit Criteria	- A fully reviewed source program - One or more Code Review checklists for every program reviewed - All identified defects fixed - Completed Time and Defect Recording logs

Finally, the Postmortem Script, shown in Table 6, describes the activities of the Postmortem phase, which includes an assessment of both process and product and an analysis of the injected defects, noting the phases at which they were removed. Analyzing the process and understanding where and why mistakes are committed allows developers to improve their own processes and outputs.

Table 6: Postmortem Script

Postmortem Script

Purpose	To guide the PSP postmortem process
Entry Criteria	- Problem description and requirements statement - Project Plan Summary form with program size, development time, and defect data - For projects lasting several days or more, completed Task Planning and Schedule Planning templates - Completed Test Report template - Completed Design templates - Completed Design Review and Code Review checklists - Completed Time and Defect Recording logs - A tested and running program that conforms to the coding and size counting standards

Step	Activities	Description
1	Defect Recording	- Review the Project Plan Summary to verify that all of the defects found in each phase were recorded. - Using your best recollection, record any omitted defects.
2	Defect Data Consistency	- Check that the data on every defect in the Defect Recording log is accurate and complete.

		- Verify that the numbers of defects injected and removed per phase are reasonable and correct. - Determine the process yield and verify that the value is reasonable and correct. - Using your best recollection, correct any missing or incorrect defect data.
3	Size	- Count the size of the completed program. - Determine the size of the base, deleted, modified, base additions, reused, new reusable code, and added parts. - Enter these data in the Size Estimating template. - Determine the total program size. - Enter this data in the Project Plan Summary form.
4	Time	- Review the completed Time Recording log for errors or omissions. - Using your best recollection, correct any missing or incomplete time data.

Exit Criteria	- A thoroughly tested program that conforms to the coding and size counting standards - Completed Design templates - Completed Design Review and Code Review checklists - Completed Test Report template - Completed Project Plan Summary form - Completed PIP forms describing process problems, improvement suggestions, and lessons learned - Completed Time and Defect Recording logs

3 Formal Methods

Formal methods hold fast to the tenet that programs should be *proven* to satisfy their specifications. Proof is the mathematical activity of arriving at knowledge deductively, starting with postulated, supposed, or self-evident principles and performing successive inferences, each of which extracts a conclusion out of previously arrived-at premises.

In applying this practice to programming, the first principle is the semantics of programs. Semantics allows us to understand program code and know what each part of the program actually computes. This makes it possible, in principle, to deductively ascertain that the computations carried out by the program satisfy certain properties. Among these properties are input-output relationships or patterns of behavior that constitute a precise formulation of the functional specification of the program or system at hand.

Formal logic, at least in its contemporary mathematical variety, strives to formulate artificial languages that frame the mathematical activity. According to this aim, there should be a language for expressing every conceivable mathematical proposition and also a language for expressing proofs, so that a proposition is provable in this language if and only if it is actually true. This latter desirable property of the language is called its correctness. This kind of research began in 1879 with Frege for the purpose of making it undisputable whether a proposition was correctly proven or not [Frege 1967]. Indeed, the whole point of devising artificial languages was to make it possible to automatically check whether a proposition or a proof was correctly written in the language. The proofs were to be accepted on purely syntactic (i.e., formal) grounds and, given the "good" property of correctness of the language, that was enough to ensure the truth of the asserted propositions.

Frege's own language turned out to be not correct and shortly after its failure the whole enterprise of formal logic took a different direction, shifting toward the study of artificial languages as mathematical objects in order to prove their correctness by elementary means. This new course was also destined to failure.

The overall outcome is nevertheless very convenient from an engineering viewpoint. Using the technology we now have available, we can go back to Frege's programs and develop formal proofs semi-automatically. The proof systems (or languages) are still reliable, although they are not complete (i.e., not every true proposition will be provable). But this is no harm in practice and the systems are perfectly expressive from an engineering perspective. All these advances allow us to define formal methods in software engineering as a discipline based on the use of formal languages and related tools for expressing specifications and carrying out proofs of correctness of programs.

Note that the semi-automatic process of program correctness proof is a kind of static checking. We can think of it as an extension of compilation, which not only checks syntax but also properties of functional behavior. Therefore it is convenient to employ the general idea of a semi-automatic verifying compiler to characterize the functionality of the tools employed within a formal methods framework.

Design by Contract (DbC) is a methodology for designing software based on the idea that specifications of software components arise, like business contracts, from agreements between a user and

a supplier, who establish the terms of use and performance of the components. That is to say, specifications oblige (and enable) both the user and the supplier to certain conditions of use and a corresponding behavior of the component in question.

In particular, DbC has been proposed in the framework of object-oriented design (and specifically in the language Eiffel) and therefore the software components to be considered are usually classes. The corresponding specifications are pre- and post-conditions to methods, establishing respectively their terms of use and corresponding outcomes, as well as invariants of the class (i.e., conditions to be verified by every visible state of an instance of the class). In the original DbC proposal, all the specifications were written in Eiffel and are computable (i.e., they are checkable at runtime).

Therefore, DbC in Eiffel provides at least the following:

- a notation for expressing the design that seamlessly integrates with a programming language, making it easy to learn and use
- formal specifications, expressed as assertions in Floyd-Hoare style [Hoare 1969]
- specifications checkable at runtime and whose violations may be handled by a system of exceptions
- automatic software documentation

However, DbC is not by itself an example of a formal method, as defined above. When we additionally enforce proving that the software components fit their specifications, we are using Verified Design by Contract (VDbC). This can be carried out within several environments, all of which share the characteristics mentioned above, including the following:

- Java Modeling Language (JML) implements DbC in Java. VDbC can then be carried out using tools like Extended Static Checking (ESC/Java) [Cok 2005] or TACO [Galeotti 2010].
- Perfect Developer [Crocker 2003] is a specification and modeling language and tool which, together with the Escher C Verifier allow performing VDbC for C and C++ programs.
- Spec# [Barnett 2004] allows VDbC within the C# framework.
- Modern Eiffel [Eiffel 2012] within the Eiffel framework.

4 Adaptation

In this section we describe the $PSP_{VDC,}$ which introduces new phases as well as modifying others already present in the ordinary PSP. In each case we describe in detail the corresponding activities and show the new scripts. Figure 2 shows the PSP_{VDC}. We assume the engineer will be using an environment similar to those listed at the end of the previous section, meaning that a computerized tool (akin to a verifying compiler) is used for

- checking the syntax of formal assertions. These are written in the language employed in the environment (e.g., as Java Boolean expressions, if JML is used) which we call the *carrier* language.
- computing proof obligations (i.e., given code with assertions, to establish the list of conditions that need to be proven in order to ascertain the correctness of the program)
- developing proofs in a semi-automatic way

The elements of Figure 2 described below summarize the most relevant novelties of PSP_{VDC}.

- After the Design Review phase, a new phase of Test Case Construct is added. This phase is used to determine the set of test cases to use in the validation of the program.
- After the Test Case Construct phase, a new phase called Formal Specification is added. In this phase the design is formalized, in the sense that class invariants and pre- and post-conditions to methods are made explicit and formal (in the carrier language).
- The Formal Specification Review is used to detect defects injected in the formal specification produced in the previous phase. A review script is used for this activity.
- The Formal Specification Compile phase consists of automatically checking the formal syntax of the specification.
- The Pseudo Code phase consists of writing down the pseudo code of every method.
- The Pseudo Code Review phase consists of precisely reviewing the pseudo code produced in the former phase.
- The Proof phase comes after production, review, and compilation of the code. The general idea is to supplement the design with formal specifications of the components and the code with a formal proof to show that it matches the formal specifications. This proof is to be carried out with the help of a tool akin to a verifying tool, in the sense that it is employed to statically check the logical correctness of the code besides its syntactic well-formedness.

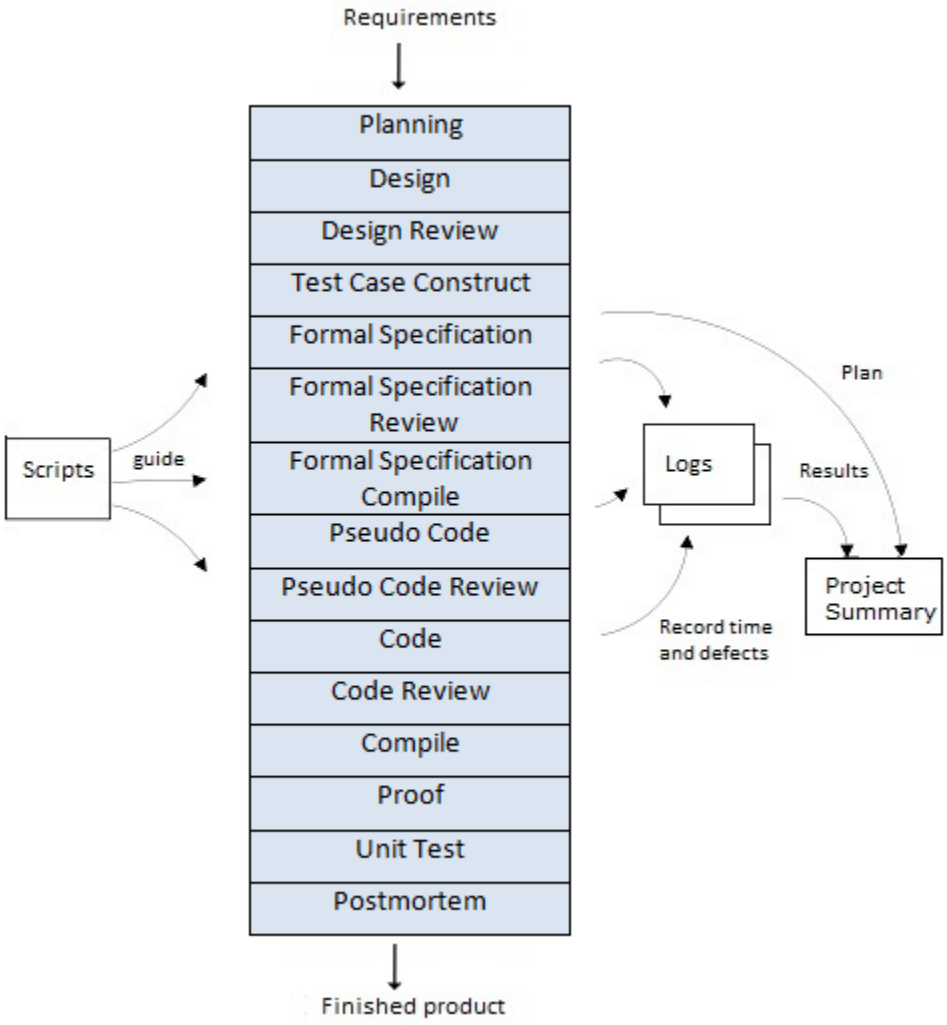

Figure 2: Phases of the PSP$_{VDC}$

In the following subsections we present in detail all the phases of the PSP$_{VDC}$, indicating in each case the activities to be performed and the modifications introduced in the scripts with respect to the original PSP.

4.1 Planning

The activities of the Planning phase in PSP$_{VDC}$ are the same as in ordinary PSP: Program Requirements, Size Estimate, Resource Estimate, Task and Schedule Planning, and Defect Estimate.

Program Requirements is for ensuring a precise understanding of every requirement. This activity is the same as in the ordinary PSP.

Size Estimate involves carrying out a conceptual design (i.e., producing a module (class) structure). Each class is refined into a list of methods and the relative size of the methods of each class is estimated. This is done in the same way as in ordinary PSP: using proxies to create a categorization of the method according to its size and the functional type of the corresponding class. Categories of

size of methods include very small, small, medium, large, or very large; functional kinds of classes include Calc, Logic, IO, Set-Up, and Text. Thus, using the structure of classes, the number of methods (and the size) in each class and the category of the class, we arrive at an estimation of the LOCs of the program.

PSP uses LOCs for estimating the size of the program and deriving from that an estimation of the effort required for its construction. It has been established that under certain conditions the effort is proportional to the LOCs of the program [Humphrey 05]. PSP_{VDC} requires engineers to formally write down the pre- and post-conditions of each method and the invariant of each class, which is a kind of output akin to LOCs and could certainly increase the total cost of development. Nevertheless, we also continue measuring the size of the product in LOCs and postulate that the relationship between effort and size in LOCs will keep valid. It will depend on the outcome of empirical studies whether we should adjust this premise and consider also Lines of Formal Specification (LOFs) for effort estimation. Note that, for estimating LOFs, it will be necessary to specify what exactly a LOF is, which will give a criterion for counting them. It will also be necessary to use a proxy for LOF estimation. The development of the corresponding techniques is out of the scope of the present work. Because of these considerations, the activity of size estimate remains unchanged in PSP_{VDC}.

Resource Estimate estimates the amount of time needed to develop the program. For this, the PROBE method is used, which employs historical records and linear regression for producing the new estimation and for measuring and improving the precision of the estimations. In our adaptation, the activity remains conceptually the same, but will employ records associated to the new phases incorporated into PSP_{VDC}. Therefore, once sufficient time data has been gathered, we will be able to estimate the effort (measured in minutes) required for the formal specification and for the program proof.

Task and Schedule Planning is for long-term projects. These are subdivided into tasks and the time is estimated for each task. This is unchanged in PSP_{VDC}.

Defect Estimate Base is for estimating the number of defects injected and removed at each phase. Historical records and the estimated size of the program are utilized for performing this estimation. In PSP_{VDC} new records are needed to estimate the defects removed and injected at each new phase.

Finally, the Planning Script in PSP_{VDC} is the same as in PSP, given that the corresponding activities are unchanged.

4.2 Design

During Design, the data structures of the program are defined, as well as its classes and methods, interfaces, components, and the interactions among all of them. In PSP, elaboration of the pseudo code is also included. In PSP_{VDC} the elaboration of the pseudo code is postponed until the formal specification is available for each method. Therefore, we eliminate from the Design phase the use of the Logic Template, which corresponds to the pseudo code. The Logic Template ceases to be a member of the set of templates of the Design Template, given that in PSP_{VDC} it is not a design template anymore.

Normally, although not specified in PSP, the Design phase also includes the design of the test cases. In PSP_{VDC} we propose a test case design in a phase separate from the Design phase because we

are interested in getting information about the time employed specifically in the construction of test cases. As explained below, such knowledge will be useful in comparing the cost of using formal methods versus that of testing and debugging.

Formal specification of methods and of invariants of classes could be carried out within the Design phase. This, however, does not allow us to keep records of the time employed specifically in Design as well as in Formal Specification. Instead, we would just record a likely significant increase in Design time. Therefore we prefer to separate the phase of Formal Specification.

The changes to process scripts appear in red text; deletions are marked with strikethrough. In sum, the activity of Design within the Development Script is modified to

Step	Activities	Description
1	Design	- Review the requirements and produce an external specification to meet them. - Complete Functional and Operational Specification templates to record this specification. - Produce a design to meet this specification. - Record the design in Functional, Operational, and State, ~~and Logic Specification~~ templates. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.

4.3 Design Review

This is the same as in ordinary PSP and uses its Development Script.

4.4 Test Case Construct

We want to investigate the cost effectiveness of test case construction and unit testing when formal methods are used. That is, is it practical to eliminate the Unit Test phase when using these formal methods? To answer this, we need to know

- The cost of test case construction
- The cost of unit test execution
- The defect density entering into unit test
- The yield of the unit test phase

This will also allow us to assess the economic and quality benefits of implementing VDbC using PSP. The Test Case Construct activity is incorporated into the Development Script as detailed below:

Step	Activities	Description
3	Test Case Construct	- Design test cases and record them in the Test Report. - Record time in the Time Recording log.

4.5 Formal Specification

This phase must be performed after Design Review. The reason for this is that reviews are very effective in detecting defects injected during design and we want to discover them as early as possible.

In this phase we start to use the computerized environment supporting VDbC. Two activities are carried out in this phase: Construction and Specification. Construction consists of preparing the computerized environment and defining within it each class with its method headers. If this is instead be done during Design as part of the functional template, omit it here. The choice is a personal one.

The second activity is Specification, in which we write down in the carrier language the pre- and post-conditions of each method as well as the class invariant. Note that, within the present approach, the use of formal methods begins once the design has been completed. It consists of the formal specification of the produced design and the formal proof that the final code is correct with respect to this specification.

Formal Specification is incorporated into the Development Script. A standard template for the specification is used in this activity. Table 7 presents an example for the language JML.

Table 7: Formal Specification Standard Template

Step	Activities	Description
4	Formal Specification	- Implement the design following the Formal Specification standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.

Purpose	To guide the formal specification of programs
Program Headers	Begin all programs with a descriptive header. The header should use the Java documentation commenting convention ("/**") so automated documentation generation is possible. Include in the descriptive header the name of the author who writes the formal specification and a version number .
Header Format	/** * @formal specification author Philip Johnson * @formal specification version Tue Dec 26 2011 */
Identifiers	Use descriptive names for all variables, constants, and other identifiers. Avoid abbreviations or single letter variables.

Identifier Example	//@ public constraint age >= \old(age); //this is good
	//@ public constraint i >= \old(i); //this is bad

Comments	Document the code so that the reader can understand its operation. Comments should explain both the purpose and behavior of the code. Comment variable declarations to indicate their purpose.

Good Comment	/*@ requires array != null; @ ensures (* return the sum of the array elements *) @ && \result == (\sum int I; 0 <= I && I < array.length; array[I]); @ ensures (* without modifying the array *) @ && (\forall int I; 0 <= I && I < array.length; @ array[I] == \old(array[I])); @*/
Bad Comment	This comment is wrong: /*@ @ (* comment *) assertion @*/ This comment is OK: /*@ @ (* comment *) && assertion @*/ Comments are treated as assertions; therefore, they should be connected to other assertions by means of &&.
Indenting	Indent every level of brace from the previous one.
Indenting Example	/*@ public normal_behavior @ requires divisor > 0; @ ensures divisor*\result <= dividend @ && divisor*(\result+1) > dividend; @ @ also @ public normal_behavior @ requires divisor == 0; @ ensures \result == 0; @*/
Capitalization	• Always use lower case in variable declarations. • Use upper case for types and clases. • Use upper case in invocations of a method so declared or of a JML library.
Capitalization Example	/*@ public model String name; @ public represents name <- getName(); @ @ public invariant !"".equals(name); */

4.6 Formal Specification Review

Using a formal language for specifying conditions is not a trivial task, and both syntactic and semantic defects can be injected. To avoid the propagation of these errors to further stages and the resulting increase in the cost of correction, we propose a phase called Formal Specification Review.

The script that corresponds to this phase contains these activities: Review, Correction, and Checking. The Review activity consists of inspecting the sentences of the specification using a checklist. In the Correction activity, all defects detected during Review are removed. Finally, Checking consists of looking over the corrections to verify their adequacy.

The Formal Specification Review activity is incorporated into the Development Script; the Formal Specification Review Script and Formal Specification Review Checklist are proposed for use in this activity.

Table 8: Specification Review Script, PSP_{VDC}

Step	Activities	Description
5	Formal Specification Review	- Follow the Formal Specification Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.

Purpose	To guide you in reviewing detailed designs
Entry Criteria	Specification Review checklist Defect Type standard Time and Defect Recording logs
General	Where the Specification was previously verified, check that the analyses covered all of the Specification, were updated for all Specification changes, and are clear and complete.

Step	Activities	Description
1	Preparation	Examine the specification and checklist and decide on a review strategy.
2	Review	Follow the Specification Review checklist. Review the entire specification for each checklist category; do not try to review for more than one category at a time! Check off each item as you complete it. Complete a separate checklist for each product or product segment reviewed.
3	Fix Check	Check each defect fix for correctness. Re-review all changes. Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space.

Exit Criteria	A fully reviewed detailed Specification One or more Specification Review checklists for every specification reviewed Documented Specification analysis results All identified defects fixed and all fixes checked Completed Time and Defect Recording logs

Table 9: Formal Specification Review Checklist Template

Student		Date	
Program		Program #	
Instructor		Language	
Formal Specification Language			

Purpose	To guide you in conducting an effective specification review
General	Review the entire specification for each checklist category; do not attempt to review for more than one category at a time! As you complete each review step, check off that item in the box at the right. Complete the checklist for one specification or specification unit before reviewing the next.
General	To verify that the formal specification adequately complements the design

Assertions	Assertions are prefixed by //@ or appear between /*@ … @*/ Every assert clause must end in ;. Verify that the variable associated to each clause \forall, \sum, \exists, etc. is appropriately initialized. In each clause \forall, \sum, \exists, etc. verify balance of parentheses in IF, ELSE, FOR, WHILE. In each clause \forall, \sum, \exists, etc. verify that the appropriate segment of the array is traversed. Verify that every method invoked within an assertion is declared as /*@ pure @*/.				
Preconditions	Method preconditions are declared by means of the requires clause.				
Postconditions	Method postconditions are declared by means of the ensures clause.				
Class Invariants	Class invariants are declared by means of the invariant clause.				

4.7 Formal Specification Compile

Any computerized tool supporting VDbC will be able to compile the formal specification. Since this allows an early detection of errors, we consider it valuable to explicitly introduce this phase into PSP_{VDC}. In particular, it is worthwhile to detect all possible errors in the formal specifications before any coding is carried out. A further reason to isolate the compilation of the formal specification is to allow the time spent in this specific activity to be recorded.

The activity Formal Specification Compile is added to the Development Script.

Step	Activities	Description
6	Formal Specification Compile	- Compile the formal specification until there are no compile errors. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.

4.8 Pseudo Code

The Pseudo Code phase allows us to understand and structure the solution to the specified problem just before coding. The pseudo code of each class method defined in the Logic Template is written down.

We propose that the pseudo code be produced after the compilation of the specification in order for the specification to serve as a well understood starting point for design elaboration in pseudocode. Writing down the pseudo code just before coding allows us to follow a well-defined process in which the output of each stage is taken as input to the next one.

The activity Pseudo Code is incorporated into the Development Script.

Step	Activities	Description
7	Pseudo Code	- Produce a Pseudo Code to meet the design. - Record the Design Logic Specification templates. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.

4.9 Pseudo Code Review

A check list is used for guiding the activity in this phase. The activity Pseudo Code Review is added to the Development Script. The Pseudo Code Review script is proposed for use in this activity.

Step	Activities	Description
8	Pseudo Code Review	- Follow the Pseudo Code Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.

4.10 Code, Code Review, and Code Compile

Just as in ordinary PSP, these phases consist of translating the design into a specific programming language, revising the code, and compiling it. The descriptions of these activities in the PSP_{VDC} Development Script are the same as in the PSP Development Script.

4.11 Proof

This phase is added in PSP_{VDC} to provide evidence of the correctness of the code with respect to the formal specification (i.e., its formal proof). A computerized verifying tool is used which derives proof obligations and helps to carry out the proofs themselves.

The description of the activity Proof within the Development Script is as follows.

12	Proof	- Construct a formal proof of correctness of the code with respect to the formal specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.

4.12 Unit Test

This phase is the same as in ordinary PSP. We consider it relevant for detecting mismatches with respect to the original, informal requirements of the program. These defects can arise at several points during the development, particularly as conceptual or semantic errors of the formal specifications. The test cases to be executed must therefore be designed right after the requirements are established (i.e., during the phase Test Case Construct) as already indicated.

The description of this activity in the PSP_{VDC} Development Script is the same as in the PSP Development Script.

4.13 Post-Mortem

This is the same as in ordinary PSP and its description in the PSP_{VDC} Development Script is the same as in the PSP Development Script.

However, several modifications have to be made to the infrastructure supporting the new process. For instance, all new phases must be included in the support tool to keep track of the time spent at each phase, as well as to record defects injected, detected, and removed at each phase. Our intention in this paper is to present the changes in the process in order to incorporate VDbC. The adaptation of the supporting tools, scripts, and training courses is a matter for a separate work.

We have now completed the description of the modifications made to each phase of the PSP to turn it into PSP_{VDC}. In Table 10 we present the PSP_{VDC} Process Script. This contains some modifications due to the changes made to the Development Script. In Table 11 we present the complete PSP_{VDC} Development Script. In the Appendix, all scripts and templates of PSP_{VDC} are shown.

Table 10: Process Script, PSP_{VDC}

Purpose	To guide the development of module-level programs
Entry Criteria	- Problem description - PSP Project Plan Summary form - Size Estimating template - Historical size and time data (estimated and actual) - Time and Defect Recording logs - Defect Type, Coding, and Size Counting standards - Stopwatch (optional)

Step	**Activities**	**Description**
1	Planning	- Produce or obtain a requirements statement. - Use the PROBE method to estimate the added and modified size and the size prediction interval of this program. - Complete the Size Estimating template. - Use the PROBE method to estimate the required development time and the time prediction interval. - Complete a Task Planning template. - Complete a Schedule Planning template. - Enter the plan data in the Project Plan Summary form. - Complete the Time Recording log.
2	Development	- Design the program. - Document the design in the design templates. - Review the design, and fix and log all defects found. - Design the test cases. - Formally specify all methods of the classes introduced in design.

		- Review formal specification and fix and log all defects found. - Compile formal specification and fix and log all defects found. - Write down pseudo code using the Logic Template. - Review pseudo code and fix and log all defects found. - Implement the design. - Review the code and fix and log all defects found. - Compile the program and fix and log all defects found. - Construct formal proof of correctness of code with respect to its formal specification. - Test the program and fix and log all defects found. - Complete the Time Recording log.
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data.

Exit Criteria	- A thoroughly tested program - Completed Project Plan Summary form with estimated and actual data - Completed Size Estimating and Task and Schedule Planning templates - Completed Design templates and Formal Specification Templates - Project or other processing unit containing formal proof of code correctness. (This depends on the concrete computerized tool employed.) - Completed Design Review, Formal Specification Review, Pseudo Code Review and Code Review checklists - Completed Test Report template - Completed PIP forms - Completed Time and Defect Recording logs

Table 11: Development Script, PSP_{VDC}

Purpose	To guide the development of small programs
Entry Criteria	- Requirements statement - Project Plan Summary form with estimated program size and development time - For projects lasting several days or more, completed Task Planning and Schedule Planning templates - Time and Defect Recording logs - Defect Type standard and Coding standard

Step	Activities	Description
1	Design	- Review the requirements and produce an external specification to meet them. - Complete Functional and Operational Specification templates to record this specification. - Produce a design to meet this specification. - Record the design in Functional, Operational, State, ~~and Logic Specification~~ templates. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
2	Design Review	- Follow the Design Review script and checklist and review the design. - Fix all defects found.

		- Record defects in the Defect Recording log. - Record time in the Time Recording log.
3	Test Case Construct	- Design test cases and record them in the Test Report. - Record time in the Time Recording log.
4	Formal Specification	- Implement the design following the Formal Specification standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
5	Formal Specification Review	- Follow the Formal Specification Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Formal Specification Compile	- Compile the formal specification until there are no compile errors. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.
7	Pseudo Code	- Produce a Pseudo Code to meet the design. - Record the Design Logic Specification templates. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.
8	Pseudo Code Review	- Follow the Pseudo Code Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
9	Code	- Implement the design following the Coding standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
10	Code Review	- Follow the Code Review script and checklist and review the code. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
11	Compile	- Compile the program until there are no compile errors. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
12	Proof	- Construct formal proof of correctness of the code with respect to its formal specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
13	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log. - Complete a Test Report template on the tests conducted and the results obtained.

Exit Criteria	- A thoroughly tested program that conforms to the Coding standard - A formal specification conforming to the Formal Specification standard

	- Completed Design and Formal Specification templates - Completed Design Review, Pseudo Code Review, Formal Specification Review, and Code Review checklists - Completed Test Report template - Completed Time and Defect Recording logs

5 Quality Planning

Quality planning in PSP includes the following:

- estimating the total number of defects injected and removed
- estimating the number of defects injected and removed at each phase
- estimating the time required at each phase

In this section we present the modifications to Quality Planning introduced in PSP_{VDC}.

For estimating the total number of defects injected, PSP uses the estimation of the size of the program as well as historical data about the amount of defects injected per KLOC. For estimating the number of defects injected and removed at each phase, PSP performs a distribution of the total estimate, making use of historical data.

In PSP_{VDC} the new phases must be taken into account in order to perform the corresponding estimations of the number of defects and of the time required. Initially, the corresponding historical data mentioned above is not available. Therefore, the initial estimation must be done by applying expert judgment. After performing several studies, accumulated data is available for employment in the desired estimations.

In PSP some benchmarks are known that also can be used for estimating the number of defects removed. In particular, from the PSP data the following rates of defect removal are known, which usually indicate good use of the process:

- 3 to 5 defects per hour in design review
- 5 to 10 defects per hour in code review

Eventually PSP_{VDC} use will produce useful benchmarks for the Formal Specification Review and Pseudo Code Review phases.

In PSP the Process Quality Indicator (PQI) suggests the following values for code and design reviews:

- the time employed in design review is not less than 50% of the time employed in design
- the time employed in the code review is not less than 50% of the time employed in coding

We are interested in obtaining, by empirical means, a relationship between the time required by the formal specification and that required by its review. Similar information is desired for the pseudo code.

6 Quality Measures

Product quality is an essential issue in PSP. Developers must remove defects, determine the causes of their injection, and learn to prevent them from occurring. PSP proposes reviews as a recommended method for defect removal because it is even more effective than testing. [Hayes1997, Vallespir 11, Vallespir 12]. To perform efficient reviews it is necessary to make measurements [Gilb 1993].

PSP defines several measurements of process quality and control, including the following:

- yield
- defect removal efficiency
- defect removal leverage
- cost of quality (COQ)

The yield of a phase is defined as the percentage of defects found at the phase in question over the total number of defects that enter the phase. It is usually employed for measuring the effectiveness of design and code reviews, as well as of compilation and testing. It can be used in PSP_{VDC} for measuring the effectiveness of the new phases of formal specification review (FSR) and pseudo code review (PCR), formal specification, compile, and proof.

The yield of the process is calculated as the percentage of defects injected and removed prior to the first code compilation. In PSP_{VDC}, this must be adjusted by taking into account the new phases that precede the compilation phase.

$$\text{Yield (process)} = 100 \cdot \frac{\text{Defects removed before code compile}}{\text{Defects injected before code compile}}$$

Defect removal efficiency is the number of defects removed per hour at the phases of Design Review, Code Review, Compile, and Test. In PSP_{VDC} it is important to also know the number of defects removed per hour in the phases of Formal Specification Review, Pseudo Code Review, Formal Specification Compile (FSC), and Proof (PRF). Defect removal efficiency for such cases is defined as follows:

$$\text{Defect removal efficiency (FSR)} = 60 \cdot \frac{\text{Defects removed in FSR}}{\text{Time in FSR (minutes)}}$$

$$\text{Defect removal efficiency (FSC)} = 60 \cdot \frac{\text{Defects removed in FSC}}{\text{Time in FSC (minutes)}}$$

$$\text{Defect removal efficiency (PCR)} = 60 \cdot \frac{\text{Defects removed in PCR}}{\text{Time in PCR (minutes)}}$$

$$\text{Defect removal efficiency (PRF)} = 60 \cdot \frac{\text{Defects removed in PRF}}{\text{Time in PRF (minutes)}}$$

Defect removal leverage is the number of defects removed per hour at one stage of the process with respect to a base phase. Normally, the base phase is Unit Test (UT). In PSP_{VDC} we propose to incorporate the indicators DRL (FSR/UT), DRL (PCR/UT), DRL (FSC/UT), and DRL (PRF/UT), which correspond to the number of defects per hour removed at FSR, PCR, FSC, and PRF respectively, with respect to the UT phase.

Cost of quality (COQ) is a measure of process quality. The components of COQ are failure, appraisal, and prevention costs. Failure cost is the time dedicated to repair and re-work, which corresponds in PSP to the phases of Compile and Test. Appraisal cost is the time spent in inspection, which in PSP is the time spent at the phases of Design and Code Review. Defect prevention is the time dedicated to the identification and resolution of the causes of the defects.

With the same idea, in PSP_{VDC} failure cost corresponds to the time employed in the phases of Code Compilation, Formal Specification Compile, Proof, and Test. The appraisal cost, on the other hand, is the time spent at the phases of Design and Code Review, Formal Specification Review, and Pseudo Code Review.

The indicator Appraisal Cost of Quality (% Appraisal COQ) is defined in PSP as the percentage of the total development time employed in design and code review. High values of this indicator are associated to low number of defects in testing and high quality of the product. We modify this indicator in PSP_{VDC} in order to incorporate the time employed in review of the formal specification and of the pseudo code. Therefore, the corresponding formula becomes

$$\%\ \text{Appraisal COQ} = 100 \cdot \frac{\text{Design Review Time} + \text{Code Review Time} + \text{FSR Time} + \text{PCR Time}}{\text{Total Development Time}}$$

The indicator Percent Failure COQ (% Failure Cost of Quality) is defined in PSP as the percentage of the total development time employed in compilation and testing. We modify it in PSP_{VDC} in order to incorporate the time spent in compilation of the formal specification (FSC) and the time spent in making the Proof. We thus rewrite the formula as

$$\%\ \text{Failure COQ} = 100 \cdot \frac{\text{Code Compile Time} + \text{Test Time} + \text{FSC Time} + \text{Proof Time}}{\text{Total Development Time}}$$

A useful COQ measurement is the rate between appraisal and failure costs (A/FR). This indicator is only implicitly modified in PSP_{VDC} because of the changes in A and FR.

In PSP, a value of A/FR greater than 2 is considered an indicator of high performance. This benchmark value must be adjusted in PSP_{VDC} after performing empirical studies because of the possible impact of the incorporated phases.

7 Conclusions and Future Work

This paper has described PSP_{VDC}, a combination of PSP with Verified Design by Contract (VDbC), with the aim of developing better quality products.

In summary, we propose to supplement the design with formal specifications of the pre- and post-conditions of methods as well as class invariants. This gives rise to seven new phases which come after the Design phase, namely Test Case Construct, Formal Specification, Formal Specification Review, Formal Specification Compile, Pseudo Code, Pseudo Code Review, and Proof. We also propose to verify the logical correctness of the code by using an appropriate tool, which we call a *verifying compiler*. This motivates the new Proof phase, which provides evidence of the correctness of the code with respect to the formal specification.

The process can be carried out within any of several available environments for VDbC.

By definition, in Design by Contract (and thereby, also in VDbC) the specification language is seamlessly integrated with the programming language, either because they coincide or because the specification language is a smooth extension of the programming language. As a consequence, the conditions making up the various specifications are Boolean expressions that are simple to learn and understand. We believe that this makes the approach easier to learn and use than the ones in other proposals [Babar 2005, Suzumori 2003]. Nonetheless, the main difficulty associated with the method resides in developing a competence in carrying out the formal proofs of the written code. This is, of course, common to any approach based on formal methods. Experience shows, however, that the available tools are generally of great help in this matter. There are reports of cases in which the tools have generated the proof obligations and discharged up to 90% of the proofs automatically [Abrial 2006].

We conclude that it is possible in principle to define a new process which integrates the advantages of both PSP and formal methods, particularly VDbC. In our future work, we will evaluate the PSP_{VDC} in actual practice by carrying out measurements in empirical studies. The fundamental aspect to be measured in our evaluation is the quality of the product, expressed in the amount of defects injected and removed at the various stages of development. We are also interested in measures of the total cost of the development.

Appendix

In this section we present the Process Script, the Development Script, the Formal Specification Standard Template, the Specification Review Script, and Formal Specification Review Checklist Template.

Table 12: Process Script, PSP_{VDC}

Purpose	To guide the development of module-level programs
Entry Criteria	- Problem description - PSP Project Plan Summary form - Size Estimating template - Historical size and time data (estimated and actual) - Time and Defect Recording logs - Defect Type, Coding, and Size Counting standards - Stopwatch (optional)

Step	**Activities**	**Description**
1	Planning	- Produce or obtain a requirements statement. - Use the PROBE method to estimate the added and modified size and the size prediction interval of this program. - Complete the Size Estimating template. - Use the PROBE method to estimate the required development time and the time prediction interval. - Complete a Task Planning template. - Complete a Schedule Planning template. - Enter the plan data in the Project Plan Summary form. - Complete the Time Recording log.
2	Development	- Design the program. - Document the design in the design templates. - Review the design and fix and log all defects found. - Design Test cases. - Formally specify the methods of every class introduced at design. - Review the formal specification and fix and log all defects found. - Compile the formal specification and fix and log all defects found. - Write down the pseudo code, using the Logic Template. - Review the pseudo code and fix and log all defects found. - Implement the design. - Review the code and fix and log all defects found. - Compile the program and fix and log all defects found. - Construct a formal proof of correctness of the code with respect to its formal specification. - Test the program and fix and log all defects found. - Complete the Time Recording log.
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data.

Exit Criteria	- A thoroughly tested program - Completed Project Plan Summary form with estimated and actual data - Completed Size Estimating and Task and Schedule Plan-

	ning templates - Completed Design templates and Formal Specification Templates - Completed Design Review, Formal Specification Review, Pseudo Code Review, and Code Review checklists - Completed Test Report template - Completed PIP forms - Completed Time and Defect Recording logs

Table 13: Development Script, PSP_{VDC}

Purpose	To guide the development of small programs
Entry Criteria	- Requirements statement - Project Plan Summary form with estimated program size and development time - For projects lasting several days or more, completed Task Planning and Schedule Planning templates - Time and Defect Recording logs - Defect Type standard and Coding standard

Step	**Activities**	**Description**
1	Design	- Review the requirements and produce an external specification to meet them. - Complete Functional and Operational Specification templates to record this specification. - Produce a design to meet this specification. - Record the design in Functional, Operational, and State templates. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
2	Design Review	- Follow the Design Review script and checklist and review the design. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
3	Test Case Construct	- Design test cases and record them in the TestReport. - Record time in the Time Recording log.
4	Formal Specification	- Implement the design following the Formal Specification standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
5	Formal Specification Review	- Follow the Formal Specification Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Formal Specification Compile	- Compile the formal specification until there are no compile errors. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.
7	Pseudo Code	- Produce a Pseudo Code to meet the design. - Record the design Logic Specification templates. - Record in the Defect Recording log any defects found.

		- Record time in the Time Recording log.
8	Pseudo Code Review	- Follow the Pseudo Code Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
9	Code	- Implement the design following the Coding standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
10	Code Review	- Follow the Code Review script and checklist and review the code. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
11	Compile	- Compile the program until there are no compile errors. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
12	Proof	- Construct a formal proof of correctness of the program with respect to the formal specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
13	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log. - Complete a Test Report template on the tests conducted and the results obtained.

Exit Criteria	- A thoroughly tested program that conforms to the Coding standard - A formal specification conforming to the Formal Specification Standard - Completed Design and Formal Specification templates - Completed Design Review, Pseudo Code Review, Formal Specification Review and Code Review checklists - Completed Test Report template - Completed Time and Defect Recording logs

Table 14: Formal Specification Standard Template, PSP_{VDC}

Purpose	To guide the formal specification of programs
Program Headers	Begin all programs with a descriptive header. The header should use the Java documentation commenting convention ("/**") so automated documentation generation is possible. Include in the descriptive header the name of the author who writes the formal specification and a version number.

Header Format	/** * @formal specification author Philip Johnson * @formal specification version Tue Dec 26 2011 */
Identifiers	Use descriptive names for all variables, constants, and other identifiers. Avoid abbreviations or single letter variables.

Identifier Example	//@ public constraint age >= \old(age); //this is good
	//@ public constraint i >= \old(i); //this is bad

Comments	Document the code so that the reader can understand its operation. Comments should explain both the purpose and behavior of the code. Comment variable declarations to indicate their purpose.
Good Comment	/*@ requires array != null; @ ensures (* return the sum of the array elements *) @ && \result == (\sum int I; 0 <= I && I < array.length; array[I]); @ ensures (* without modifying the array *) @ && (\forall int I; 0 <= I && I < array.length; @ array[I] == \old(array[I])); @*/
Bad Comment	This comment is wrong: /*@ @ (* comment *) assertion @*/ This comment is OK: /*@ @ (* comment *) && assertion @*/ Comments are treated as assertions; therefore, they should be connected to other assertions by means of &&.
Indenting	Indent every level of brace from the previous one.

Indenting Example	/*@ public normal_behavior @ requires divisor > 0; @ ensures divisor*\result <= dividend @ && divisor*(\result+1) > dividend; @ @ also @ public normal_behavior @ requires divisor == 0; @ ensures \result == 0; @*/
Capitalization	• Always use lower case in variable declarations. • Use upper case for types and classes. • Use upper case in invocations of a method so declared or of a JML library.
Capitalization Example	/*@ public model String name; @ public represents name <- getName(); @ @ public invariant !"".equals(name); */

Table 15: Specification Review Script, PSP$_{VDC}$

Purpose	To guide you in reviewing detailed designs
Entry Criteria	- Specification Review checklist - Defect Type standard - Time and Defect Recording logs
General	Where the Specification was previously verified, check that the analyses covered all of the Specification, were updated for all Specification changes, and are clear and complete.

Step	**Activities**	**Description**
1	Preparation	Examine the program and checklist and decide on a review strategy.
2	Review	- Follow the Specification Review checklist. - Review the entire program for each checklist category; do not try to review for more than one category at a time! - Check off each item as you complete it. - Complete a separate checklist for each product or product segment reviewed.

3	Fix Check	- Check each defect fix for correctness. - Re-review all changes. - Record any fix defects as new defects and, where you know the defective defect number, enter it in the fix defect space.

Exit Criteria	- A fully reviewed detailed Specification - One or more Specification Review checklists for every design reviewed - Documented Specification analysis results - All identified defects fixed and all fixes checked - Completed Time and Defect Recording logs

Table 16: Formal Specification Review Checklist Template

Student		Date	
Program		Program #	
Instructor		Language	
Formal Specification Lenguage			

Purpose	To guide you in conducting an effective specification review				
General	Review the entire Specification for each checklist category; do not attempt to review for more than one category at a time! As you complete each review step, check off that item in the box at the right. Complete the checklist for one specification or specification unit before reviewing the next.				
General	To verify that the formal specification adequately complements the design.				
Assertions	Assertions are prefixed by //@ or appear between /*@ … @*/ Every assert clause must end in ;. Verify that the variable associated to each clause \forall, \sum, \exists, etc. is appropriately initialized. In each clause \forall, \sum, \exists, etc. verify balance of parentheses in IF, ELSE, FOR, WHILE. In each clause \forall, \sum, \exists, etc. verify that the appropriate segment of the array is traversed. Verify that every method invoked within an assertion is declared as /*@ pure @*/.				
Preconditions	Method preconditions are declared by means of the requires clause.				
Postconditions	Method post conditions are declared by means of the ensures clause.				
Class Invariants	Class invariants are declared by means of the invariant clause.				

References/Bibliography

URLs are valid as of the publication date of this document.

[Abrial 2006]
Abrial, Jean-Raymond. "Formal Methods in Industry: Achievements, Problems, Future*,"* 761-768. *ICSE '06: Proceedings of the 28th International Conference on Software Engineering.* Shanghai, China, May 2006. www.irisa.fr/lande/lande/icse-proceedings/icse/p761.pdf.

[Babar 2005]
Babar, Abdul and Potter, John. "Adapting the Personal Software Process (PSP) to Formal Methods," 192-201. *Proceedings of the Australian Software Engineering Conference (ASWEC'05)*. Brisbane, Australia, Mar./Apr. 2005. IEEE Computer Society Press, 2005.

[Barnett 2004]
Barnett, Mike, Rustan, K., Leino, M., and Schulte, Wolfram. "The Spec# Programming System: An Overview," 49-69. *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices.* Marseille, France, Mar. 2004. Springer-Verlag, 2004.

[Cok 2005]
Cok, David and Kiniry, Joseph. "ESC/Java2: Uniting ESC/Java and JML." *Lecture Notes in Computer Science 3362* (2005): 108-128.

[Crocker 2003]
Crocker, David. "Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement," *Lecture Notes in Computer Science 3582* (2003).

[Eiffel 2012]
*Definition of Modern Eiffel.*SourceForge, 2013. http://tecomp.sourceforge.net/index.php?file=doc/papers/lang/modern_eiffel.txt.

[Frege 1967]
Frege, G. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle a. S.: Louis Nebert, 1879. Translated as *Concept Script, a Formal Language of Pure Thought Modelled Upon That of Arithmetic*, by S. Bauer-Mengelberg in J. vanHeijenoort (ed.), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1967.

[Galeotti 2010]
Galeotti, Juan, Rosner, Nicolás, Pombo, López, and Frias, Marcelo F. "Analysis of Invariants for Efficient Bounded Verification," 25-36. *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010*, Trento, Italy, Jul. 2010. ACM, 2010.

[Gilb 1993]
Gilb, Tom and Graham, Dorothy. *Software Inspection*. Addison-Wesley, 1994 (ISBN 978-0201-631814).

[Hayes 1997]
Hayes, William; & Over, James. Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers (CMU/SEI-97-TR-001). Software Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/library/abstracts/reports/97tr001.cfm

[Hoare 1969]
Hoare, C.A.R. "An Axiomatic Basis for Computer Programming," *Communications of the ACM 12*, 10 (1969): 576-580.

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley, 2005.

[Humphrey 2006]
Humphrey, Watts S. *TSP: Coaching Development Teams.* Addison-Wesley, 2006.

[Kusakabe 2012]
Kusakabe, Shigeru; Omori, Yoichi; and Araki, Keijiro. "A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report," 67-75. *TSP Symposium 2012 Proceedings* (CMU/SEI-2012-SR-015). Software Engineering Institute, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/library/abstracts/reports/12sr015.cfm

[Meyer 1992]
Meyer, Bertrand. "Applying Design by Contract," *IEEE Computer 25*, 10 (October 1992): 40-51.

[Schwalbe 2007]
Schwalbe, Kathy. *Information Technology Project Management*, 5th edition. Course Technology, 2007 (978-1423901457).

[Suzumori 2003]
Suzumori, Hisayuki, Kaiya, Haruhiko, and Kaijiri, Kenji. "VDM Over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development." *Proceedings of the 27th Annual International Computer Software and Applications Conference*. Dallas, Texas, Sept. 2003. IEEE Computer Society, 2003.

[Rombach 2007]
Rombach, D., Münch, D., Ocampo, A., Watts, H., and Burton, D. "Teaching Disciplined Software Development." *Science Direct – The Journal of Systems and Software 81* (2007): 747 – 763.

[Vallespir 2011]
Vallespir, Diego and Nichols, William. "Analysis of Design Defects Injection and Removal in PSP," 19-25. *Proceedings of the TSP Symposium 2011: A Dedication to Excellence.* Atlanta, GA, Sept. 2011.

[Vallespir 2012]
Vallespir, Diego and Nichols, William. "An Analysis of Code Defect Injection and Removal in PSP," 3-20. *TSP Symposium 2012 Proceedings* (CMU/SEI-2012-SR-015). Software Engineering Institute, Carnegie Mellon University, 2012.
http://www.sei.cmu.edu/library/abstracts/reports/12sr015.cfm

[Wohlin 2000]
Wohlin, C., Runeson, P, Höst, M., Ohlsson, M. C., Regenell, B. and Wesslén, A. *Experimentation in Software Engineering.* Kluwer Academic Publishers, 2000.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	May 2013	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
PSPVDC: An Adaptation of the PSP that Incorporates Verified Design by Contract	FA8721-05-C-0003

6. AUTHOR(S)

Moreno, Silvana; Tasistro, Álvaro; Vallespir, Diego; and Nichols, William

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2013-TR-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	ESC-TR-2013-005

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

The Personal Software Process (PSP) promotes the use of careful procedures during all stages of development with the aim of increasing an individual's productivity and producing high quality final products. Formal methods use the same methodological strategy as the PSP: emphasizing care in development procedures as opposed to relying on testing and debugging. They also estab-lish the radical requirement of proving mathematically that the programs produced satisfy their specifications. Design by Contract (DbC) is a technique for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language. When appropriate techniques and tools are incorporated to prove that the components satisfy the established requirements, the method is called Verified Design by Contract (VDbC).

In this paper we present a proposal for integrating VDbC into PSP in order to reduce the amount of defects present at the Unit Testing phase, while preserving or improving productivity. The resulting adaptation of the PSP, called PSPVDC, incorporates new phases, modifies others, and adds new scripts and checklists to the infrastructure. Specifically, the phases of Formal Specifica-tion, Formal Specification Review, Formal Specification Compile, Test Case Construct, Pseudo Code, Pseudo Code Review, and Proof are added.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Formal methods, PSP, Personal Software Process, verified design by contract, software defects	53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL